Utility Types in TypeScript.

Utility types in TypeScript allow developers to construct new types based on existing ones. They let you manipulate and extend existing types.

They come in handy when we want to write type-safe code based on the existing functionalities.

Let's dive into the 5 most usefull utility types in TypeScript.

First, let's create an interface we will be using in the examples below.

```
interface User {
  firstName: string;
  lastName: string;
  age?: number;
  email: string;
  password: string;
}
```

In the example above, we have created an interface named User, with the fields **firstName, lastName, email and password** set as required, while age, indicated by the question mark, is optional.

Our first ulity type will be:

Pick:

```
type UserPreview = Pick<User, "firstName" | "lastName" | "age" | "email">;

const userPreview: UserPreview = {
  firstName: "John",
  lastName: "Doe",
  age: 28,
  email: "john.doe@gmail.com"
};
```

In the example above, we have picked selected fields from the User interface. As a result, our new type UserPreview doesn't include the password field anymore.

This is very handy to create new shorter types from the existing ones, but don't you feel that we could do it in a shorter way?

Yes, we can! This is where the Omit type comes in handy.

Omit:

```
type UserPreview = Omit<User, "password">;

const userPreview: UserPreview = {
  firstName: "John",
  lastName: "Doe",
  age: 28,
  email: "john.doe@gmail.com"
};
```

This way, we have created a new type named UserPreview based on the User interface, but we have omitted the password field.

This is a more efficient way to create a new type that excludes certain fields.

Readonly

When we fetch user data from the server and pass it to the global store, we often don't want to change it over time.

To make sure we will not store the password, let's first create a new type that omits the password field.

```
type UserNoPassword = Omit<User, "password">;
```

This way we have created a new type without the password field inside.

Now let's proceed to creating a new type that will be readonly.

```
type UserReadOnly = Readonly<UserNoPassword>;

const user: UserReadOnly = {
  firstName: "John",
  lastName: "Doe",
  age: 28,
  email: "john.doe@gmail.com"
}
```

Now, whenever someone will try to update any of the user's fields, they will end up with an error. For example**, if we try to change firstName:**

```
user.firstName = 'Jane';
```

TypeScript compiler will return an error

Error: Cannot assign to 'firstName' because it is a read-only property.

Partial

There are scenarios where you might want to create a type where all the properties are optional. This can be particularly useful when you are working with functions that update parts of an object or when you are dealing with objects that are constructed incrementally.

```
type PartialUser = Partial<User>;

const updateUser: PartialUser = {
  firstName: "John",
  email: "john.doe@gmail.com"
};
```

In the example above, we created a new type PartialUser where all properties from the User interface are optional. This is useful when you only need to work with or update a subset of properties.

Required

Conversely, there are situations where you need to ensure that all properties of a type are required. This can be useful when you want to make sure that an object is fully constructed with all necessary properties.

```
type CompleteUser = Required<User>;

const newUser: CompleteUser = {
  firstName: "John",
  lastName: "Doe",
  age: 28,
  email: "john.doe@gmail.com",
  password:
"$2y$10$MOvuRnKPu.QGy05uebE/A.W8TIKQT7s831XB9UJy25daOBgB18eEm"
};
```

In the example above, we created a new type **CompleteUser** where all properties from the User interface are required, including the age property**. This ensures that when creating a CompleteUser, no property is left undefined.**

By using these utility types — **Pick**, **Omit**, **Readonly**, **Partial**, and **Required** — you can effectively manipulate and extend your existing types to better fit your application needs.